R Class Notes

Richard I. Shrager
Mathematical and Statistical
Consulting Laboratory
Center for Information Technology
National Institutes of Health
Building 12A, Room 2003C
Bethesda MD 20892-5620
Mail stop: 5620
Telephone: (301)402-5864
Fax: (301)402-4544
E-mail: shragerr@mail.nih.gov

May 6, 2011

Contents

Chapter 1

Amenities

R is a high-level language for computing with numerical arrays and other data types. But before we get into that aspect of it, there are several features of the language that are not computing *per se*, but which do make your life easier.

1.1 Setting Up

To load R onto your computer, go to a site called CRAN, full name: Comprehensive R Archive Network. The web site is `http://cran.r-project.org`. From Google, typing CRAN will get you there. Select "Download and Install R", then follow instructions. Also on the CRAN web site is a set of notes titled "An Introduction to R", about 100 pages of terse but useful information.

1.2 Citing R in Publications

Give credit where credit is due. If R has been of considerable help in your work, the proper way to cite R in your publication is found by typing `citation()` during an R session.

1.3 Online Help

When you give a `help` command, e.g. `help(sqrt)`, or `?sqrt`, R displays a description of the function `sqrt`, including what it does, how to use it, and examples. If you want help on features that use non-standard characters, enclose the argument in quotes as in `help("[[")` or `?"[["`. The `help` display is in a separate window, so it does not interrupt the flow of your main session.

Using the `help` feature is somewhat cryptic, because the item you request must be a single item that the help command recognizes. The help command will not accept a search criterion like a phrase, as a web browser might. To find your way around the help files, proceed systematically. You can start by saying `help(help)` or just `help()`, which displays info on the help command itself. A more permissive form of help is the command, e.g. `help.search("square root")`, which will search the help files for mention of the phrase `square root`. The command `library()` will provide a list of available R libraries on your system. A specific library, e.g. `base`, can be explored with the command `library(help=base)`. If the programs you need are not in the currently-installed packages, see the section below: "Installing and Loading Packages".

Some topics, e.g. function names, can be entered directly as in `help(sqrt)`. Other topics, e.g. operator symbols like `/` and reserved words like `for`, must be in quotes as in `help("for")`. Of special interest is the "See Also" section in many entries, which lists related entries you might want to seek help on. Thus, having found one entry of interest, you can let "See Also" guide you to others.

Throughout these notes, special lines will appear beginning with the word `HELP:`. The topics on that line are used, e.g. as in `help("which")`, to get more information about the matters under discussion.

```
HELP:  help()  help(help)  help.search()  library()
```

Some useful starting points in your search of the help files are those functions that test for types of data. For example, the expression `is.vector(x)` will return `TRUE` if `x` is a vector and `FALSE` otherwise. There are several such functions (all beginning with `is.`). The help files on these functions can often lead to useful cross-references in their "See Also" sections. Here is a list of `is.` functions:

```
HELP:  is.array  is.character  is.complex  is.data.frame
is.double  is.factor  is.finite  is.infinite  is.list
is.logical  is.matrix  is.na  is.nan  is.numeric  is.ordered
is.raw  is.table  is.ts  is.vector
```

1.4 Online Examples

An online adjunct to `help` is `example`. Some help files are difficult to understand, because they tend to assume knowledge of R details that a new user doesn't have. To see worked examples of how a function is used, type `example(funcname)`. The number of examples thus procured usually outnumbers the examples in the `help` file.

1.5 Reference Works

If you get serious about R programming, online help will not be sufficient. Online help provides information in separate packets organized under broad topics. It does not tie the language together very well. It does not show how the various features interrelate. The most authoritative overview is:

```
Title: The New S Language
Subtitle: A Programming Environment
\ \ \ for Data Analysis and Graphics.
by Richard A. Becker, John M. Chambers, and Allan R. Wilks
```

Those interested primarily in statistical applications might also want to see:

```
Title: Statistical Models in S
Editors: John M. Chambers and Trevor J. Hastie
```

A college text on R is:

```
Title: The R Book
Author: Michael J. Crawley
Publisher: John Wiley \& Sons, Ltd.
\ \ \ West Sussex, England 2007.
(A tome, over 900 pages)
```

The “Nutshell” series has an entry on R:

```
Title: R in a Nutshell
Author: Joseph Adler
Publisher: O'Reilly Media, Sebastopol CA 2010.
```

For those users who know MATLAB, there is an excellent comparison:

```
Title: MATLAB / R Reference
Author: David Hiebeler
Source: http://www.math.maine.edu/~hiebeler
\ \ \ or Google ``MATLAB R comparison''
```

1.6 Installing and Loading Packages

Many useful programs are not included in the base package of R. For example, specialized programs for matrix analysis are in the `Matrix` package (note the initial cap), and programs for solving ordinary differential equations are in the `odesolve` package. To find out what packages are available on the CRAN web site, go to the site and click on packages. This may not be terribly helpful unless you know what you are looking for, because there are hundreds of packages. Sometimes a letter grouping provides a clue, e.g. “de” or “ode” may indicate ordinary differential equations. To install e.g. the Matrix package (place it in your library) get into R and say:

```
install.package('Matrix')
```

Each package must be installed, only once, to use it. Thereafter, to make the package available during run time, i.e. to load it, say:

```
library(Matrix)
```

1.7 Some Package Descriptions

The function `vignette` provides descriptions of some packages and some of their parts. Not all packages provide vignettes. To see some of those that do, type `vignette(all=TRUE)`, which will give you a list of vignettes in some packages that `R` knows about, even if you haven't attached them yet. If you want to see a list of vignettes in attached packages, type `vignette(all=FALSE)`. To see a particular vignette, often in pdf format, say e.g. `vignette("grid")`.

1.8 Start, Abort, Stop, and Resume

To start R, click on the R icon, or whatever convention has been set up on your computer. A window labeled "R Console" will appear. Once in R, to find out which version you are running, and its date, look at the first line displayed in the R Console.

If you create a monster R program that threatens to run forever, or that produces tons of useless output. you can abort it with the escape key, which returns control to you without quitting R. That way, you might salvage some of what you have done.

To exit R, say `quit()`, or `q()`, or click the X in the window's upper right corner. When you quit, you will be given the option to save your workspace, i.e. the names and values of all your variables. If you elect to save your workspace, it will be restored when you begin your next session.

```
HELP:  q quit
```

1.9 Functions and Scripts

A function is a code that accepts arguments and produces one output of some sort. When a function is invoked, its name is always followed by a pair of parentheses enclosing arguments if any, e.g. `sin(x)` or `quit()`.

In effect, a function can produce many outputs, but they must all be packaged in a single data structure. A function has its own workspace or frame in which its variables reside. When a function stores a value, say, `x=q`, that value does not influence any `x` that you might be using.

A script, in contrast to a function, is simply a set of commands in a file that do exactly what they would do if you typed them in directly. The variables used by the script are your variables.

See the chapter on scripts and functions.

1.10 Statement Delimiters

Statements such as `y=5*x+2` can end either with an end-of-line or with a semicolon. Statements can be continued to the next line only if it is obvious that the statement is incomplete, e.g. if a binary operator is missing an operand, or if a parenthesis is unpaired. When R expects a new statement, the prompt `>` is given. However, when R expects a continuation of the previous line, the prompt `+` is given.

1.11 Blanks and Comments

Blanks can be inserted in most reasonable places, but not within variable names, numeric symbols, or operators:

```
Delta     but not    Del ta
3.1416    but not    3 . 1 4 1 6
```

```
    %% (the mod operator) but not % %
```

Comments begin with the character `#`. Anything to the right of `#` is ignored, including other `#`s. A comment has no influence on statement continuation. If an incomplete statement is followed by a comment, R will give the `+` prompt on the next line.

1.12 Retrieving and Correcting Commands

As you type commands, mistakes can be corrected by backspacing etc. in the usual manner. Mistakes in previous commands can be corrected easily with the "up-arrow" key, which recalls previously-entered lines for editing and reentry.

1.13 Summary

The following capabilities enable and ease your R sessions:

Start, stop, or resume an R session.
Stop a runaway process.
Install libraries from remote sources.
Get off-line help from various references.
Get on-line help using `help`,
  `help.search`, and `example`.
Issue statements directly, or from scripts.
Put more than one statement on a line.
Spread one statement over several lines.
Put comments in your code.
Insert blanks in most reasonable places.
Recall previous commands for editing and reuse.

Chapter 2

Numerical Operations

2.1 Modes of Objects

Every object (data item, expression, or function) has three attributes: mode, type, and class. These attributes can be determined for an object `x` with the functions:

```
mode(x)   typeof(x)   class(x)
```

The primary modes in R are:

```
Basic Modes       Values

logical           TRUE or FALSE.
numeric           Includes the types
                     integer and double precision,
                     along with special values
                     inf, -inf, NaN, NA.
complex           Numbers with real and imaginary parts.
character         Strings of text.
raw               Bytes and bits with hex notation
                     and bitwise logical operators.
NULL              Empty data.

-------------------------------------------------------

Recursive Modes   Values

list              Data that can contain other lists.
function          Program modules,
                     possibly with arguments.
expression        Executable text.
```

Recursive objects (last 3 items) are more exotic, because they can contain objects like themselves.. Users and writers of packages may also create objects and define their attributes.

This chapter is mainly about data with a distinctly numeric flavor, namely numeric, complex, and logical. The chapter will also address the common ways of storing numbers, namely vectors, matrices, arrays, and lists.

2.2 Numbers in General

Internally, numbers are in binary 64-bit double-precision floating-point, with roughly 16 decimal places of precision, and a magnitude range of `2.2251e-308` to `1.7977e+308`. Numbers are displayed or entered as integers, or as decimal constants in either fixed point or floating point format. For example, the following are all representations of the numbers 132 (on line 1) and -0.00132 (on line 2):

```
132  132.0  1.32e2  1.32e+2  132e0
-0.00132  -1.32e-3  -132e-5
```

A sign (`+` or `-`) may appear before a constant. A sign may also appear within a constant, but only immediately following `e` in the floating point form. A real constant may not contain blanks. In complex constants, the letter `i` is suffixed to the imaginary part, as in `2+3i`. Blanks may appear between terms of an imaginary constant. As a separate entity, `1i` represents $\sqrt{-1}$. So we see that in addition to numerals, the symbols `+-.ei` may appear in numerical constants.

2.3 Exceptional Values

There are special values that numerical variables may assume, including elements of vectors and arrays (see below), and yet they are not numbers. These are:

Value	Meaning
`Inf`	`Plus infinity, generated by e.g. 1/0, 5*inf, and 10^400.`
`-Inf`	`Minus infinity, generated by e.g. -5/0, -5*Inf, and log(0).`
`NaN`	`Not-a-Number, generated by e.g. 0/0, Inf-Inf, and Inf/Inf.`
`NA`	`Not Apply, generated by deliberate definition (e.g. a default value).`

2.4 Logical Values

A logical constant is denoted TRUE or FALSE, and is usually generated by comparisons like `a<b`, or by logical operations like `a&b`. In these notes, we will sometimes use the abbreviations `T` for `TRUE` and `F` for `FALSE` for brevity, but these are not `R` conventions. You must set `T=TRUE` and `F=FALSE` if you want to use them that way. Numbers can be converted to logical using the `as.logical` function, or with the comparison e.g. `X!=0`. For any number or set of

numbers `X` of any type, `as.logical(X)` produces `F` when `x=0` and `T` otherwise. Comparisons between numbers always produces logicals, as do logical operations between logical variables. Here is a summary of logical comparisons and operations and a few common functions:

```
Comparisons:      TRUE if:
   x <  y         x is less than y.
   x <= y         x is less than or = to y.
   x == y         x is = to y.
   x != y         x is not = to y.
   x >= y         x is greater than or = to y.
   x >  y         x is greater than y.
Operations :
   x &  y         x and y are both TRUE.
   x |  y         x or y or both are TRUE.
   !x             x is FALSE.
Functions:
   all(x)         all x are TRUE.
   any(x)         any x is  TRUE.
   is.numeric(x)  x is of numeric data type.
   is.logical(x)  x is of logical data type.
   is.inf(x)      an array the same size as x
                  that contains the value TRUE
                  wherever the corresponding x is inf,
                  and FALSE otherwise.
```

2.5 NULL

The value `NULL` resides in a variable that exists, but that has no data yet, or that has had all data removed. For example, the empty comma-separated list c() is NULL. Also a newly defined list vector contains null elements, which can then be set individually, e.g.:

```
v5=vector('list',5)                # v5 contains 5 NULL elements
v5[[2]] = matrix(1:100,nrow=10)     # which can be
v5[[3]] = "Item 2 is an example of"  # individually set to
v5[[4]] = "A 10-by-10 matrix"       # various and sundry items.
```

2.6 Names of Variables

The naming conventions in R are similar to naming conventions other languages. A name may contain upper and lower case alphabetic characters, decimal digits, and the period (.) character. No spaces or operators are permitted within names.

The first character must be alphabetic or a period, but if period is first, an alphabetic must be second. Some examples are:

```
A  a  Jar23  n7Z  group.5.ages  .LaTeX.
```

Note that period is the separator of choice for readability within a name, since spaces within names are not allowed. Names are case-sensitive, so that the first two names above are distinct.

Certain words may not be used as variable names, because they are part of the grammar of R (see the chapter on grammar). These words are:

```
function for in while repeat next break if else
```

However, these words do not use upper case. Therefore, variable names like `IF` and `For` may be used without conflict.

2.7 Complex Arrays

A complex number z has a real part and an imaginary part suffixed with `i`, e.g. `z=2+3i`, or with real variables `x` and `y`, `z=x*y*1i`. There are several functions for handling z:

Function	Result
`Re(z)`	`Real part of z.`
`Im(z)`	`Imaginary part of z.`
`abs(z),Mod(z)`	`Magnitude of z.`
`Conj(z)`	`Complex conjugate of z.`
`Arg(z)`	`Argument or angle of z.`

A complex number is stored as a pair of double precision numbers, namely, the real and imaginary part, occupying 16 bytes per complex number. Operations on real and complex data look alike, which can be a bit confusing at times. For example, `sqrt(x)` does not produce a proper answer for real, negative `x`. Instead, it produces a NaN and a warning. To get a proper answer, you must express real `x` as an explicitly complex number, e.g. `sqrt(x+0i)`, or coerce x to complex as in `sqrt(as.complex(x))`, which will produce a proper complex answer.

If an array `A` is real (no imaginary parts), and you store a complex value in just one of its elements, the entire array is converted to complex format, even if the imaginary part of that single value is zero! R makes no effort to convert back to real format when imaginary parts of all entries are zero. You can do that conversion e.g. by `A=Re(A)`.

There are many R functions that are valid for both real and complex data, such as elementary functions like `sin` and `exp`, and matrix operations like `solve` and `eigen`.

```
HELP: is.comlex as.complex abs Mod Arg Conj Re Im
      sin exp solve eigen
```

2.8 Operations on Scalars

R provides several infix operators, i.e. operators like `+` that appear between operands. These are listed in the "Function Survey" chapter under "Arithmetic", along with functions for rounding. Common functions of one variable, e.g. `sin(x)`, are listed in the same chapter under "Elementary Functions". These operators and functions can also be applied to objects containing many numbers (vectors, matrices, and arrays).

2.9 Numeric Vectors

A numeric vector is an ordered list of numbers (its elements), and it has a length (the number of elements), but unlike the vectors of linear algebra, it has no row/column orientation. A vector can be formed in several ways.

```
 #  Examples:                 Resulting vectors:
 1. v  = 6                    # v gets one element.
 2. v  = c(2, 5, 3.6)         # v gets the values 2, 5, and 3.6.
 3. v2 = v                    # v2 gets the vector v.
 4. v  = c(x,y,z)             # v gets the numbers in arrays x, y, & z.
 5. v  = 3:7                  # v gets the set of values 3,4,5,6,7.
 6. v  = 6:1                  # v gets 6,5,4,3,2,1.
 7. v  = seq(2,14,3)          # v gets 2,5,8,11,14.
 8. v  = seq(0,1,len=6)       # v gets 0, .2, .4, .6, .8, 1.
 9. v  = 2^(0:4)              # v gets 1,2,4,8,16.
10. v  = rep(1:3,c(3,2,4))    # v gets 1,1,1,2,2,3,3,3,3.
11. vs = sort(v)              # vs is v in increasing order.
12. iv = order(v)             # iv is the sort-permutation index of v.
13. vs = v[iv]                # Lines 12 & 13 are another way to sort v.
```

As a special case, a scalar is a one-element vector. The function `c` creates a vector by catenating (chaining) the scalar values in its arguments, e.g. if x, y, and z in example 4 above are 3-by-2 matrices, then v becomes an 18-element vector. (A later chapter discusses what happens when the arguments to `c` are not all numeric.) Colon notation and the `seq` function create equally-spaced vectors. The functions `seq` (sequence), `rep` (repetition), `sort`, and `order` are especially useful. Consult the help files for further details and examples.

2.10 Indexing

An index is a vector (call it `k`) whose elements somehow specify the elements of another vector (call it `v`). The salient facts about indexes are these:

1. Indexes may appear in a source being accessed as in `x=v[k]` or in a target being modified as in `v[k]=x`.

2. An index may be numeric or logical.

3. A numeric index is truncated to integer so that `k=c( 1.2, 3.7 )` is regarded as `c(1,3)`.

4. Numeric indexes must be all positive or all negative.

5. A negative numeric index denotes elements that are to be ignored or omitted, so that `v[-(3:4)]` means all of `v` except elements `3` and `4`. Negative indexes are not required to refer to elements that exist, e.g. `v[-7]` could refer to `v` of length 3.

6. In a logical index, the locations of `TRUE` entries tell which elements are to be accessed in sources or altered in targets. The locations of `FALSE` entries tell which entries are to be omitted or ignored.

7. Ultimately, a logical index must have the same length as its data, e.g. in the expression `v[k]`, `length(k)` must be made to match `length(v)`.

8. If logical index `k` is initially shorter than `v`, then in `v[k]`, `k` will be extended (repeated) (see more about vector extensions below) to match `v`, so that e.g. when `k=c(TRUE,FALSE)`, the expression `v[k]` yields every odd-indexed entry in `v`.

9. Numeric indexes are more versatile than logical indexes. For any logical index, there is an equivalent numerical index, but the reverse is not true. Numerical indexes can specify repeats and permutations, where logical indexes can only select or ignore.

Here are some examples:

```
# v = c(10,20,30,40,50,60,70,80,90,100)
v = seq(10,100,len=10)
x = v[ c(3,7,5)]   # x = c(30,70,50)
x = v[-c(3,7,5)]   # x = c(10,20,40,60,80,90,100)
v[ c(3,5,7)] = 0   # v = c(10,20,0,40,0,60,0,80,90,100)
v[-c(3,5,7)] = 2   # v = c(2,2,0,2,0,2,0,2,2,2)
T=TRUE; F=FALSE    # Logical abbrevs.
k = c(T,T,F,F,T,T,F,F,T,T)
v = 10*(1:10)      # Restore v to its value at the top.
x = v[v<55]        # x = c(10,20,30,40,50) because the index
                   #    (v<55) is c(T,T,T,T,T,F,F,F,F,F)
x = v[k]           # x = c(10,20,50,60,90,100)
# Note that !k is c(F,F,T,T,F,F,T,T,F,F)
x = v[!k]          # x = c(30,40,70,80)
k = c(F,F,F,T,T)   # k is now too short for v.
# k will be cycled twice to match v.
x = v[k]           # x = c(40,50,90,100)
```

Other indexable objects (e.g. matrices, arrays, lists) are indexed with rules similar to those for vectors.

2.11 Vector Extension

In R, vectors have special privileges. All of the infix operators that work on scalars, as listed above, also work on vectors in a manner peculiar to R. If the two vectors are of equal length, the operations proceed in element-by-element fashion as in line 6 below. However, the infix operators also apply to vectors of *unequal* length, where the values of the shorter vector are repeated (rotated) to match the longer vector. In the simplest case of vector repetition, a scalar (a 1-element vector) can be added to any vector (or matrix or array) without protest, as in line 7 below. An extension works seamlessly if the longer length is an integer multiple of the shorter length, as in lines 6, 7, and 8 below. As a matter of policy, R permits arithmetic between vectors of any lengths, but warns you if the longer length is not an integer multiple of the shorter, as in line 9 below. In extreme cases, this convention can fill your session log with pointless messages. Rather than use vectors whose lengths don't divide evenly, it is often best to expand the shorter vector explicitly by coding exactly what you want it to be (lines 10 and 11 below).

```
 #  Examples                   Resulting vectors
 1. vp1 = .1                   # vp1 = scalar 0.1
 2. v3  = 1:3                  # v3  = c(1,2,3)
 3. vp3 = c(.1,.2,.3)          # vp3 = c(.1,.2,.3)
 4. vp4 = c(.1,.2,.3,.4)       # vp4 = c(.1,.2,.3,.4)
 5. v6  = 1:6                  # v6  = c(1,2,3,4,5,6)
 6. v = v3+vp3                 # v = c( 1.1,  2.2,  3.3 )
 7. v = v3+vp1                 # v = c( 1.1,  2.1,  3.1 )
 8. v = v6+vp3                 # v = c(1.1,2.2,3.3,4.1,5.2,6.3)
 9. v = v6+vp4                 # v = c(1.1, 2.2, 3.3, 4.4, 5.1, 6.2)
                               #       and a warning!
10. vp4.x = c(vp4,vp4[1:2])    # vp4.x = vp4 expanded to match v6
11. v     = v6+vp4.x           # Same as line 9, but no warning
```

2.12 Forming Rectangular Arrays

Matrices and arrays are sets of numbers (all real, all complex, or all logical) arranged in rectangular fashion. That is, they may be arranged in a row, or a column, or several rows and columns (a matrix) or a "book" of matrices, one matrix to a page (a 3-dimensional array), or a "shelf" of books (a 4-dimensional array) and so on up. Matrices and arrays have a strict format, with the same number of rows in every column, the same size matrix on every page, etc. They are created using the special functions `matrix` and `array`.

```
#  Examples                      Resulting arrays
1. r4   = matrix(c(5,3,4,2),1,4) # r4 = 5 3 4 2 (a row).
2. c4   = matrix(r4,4,1)         # c4 = r4 as a column.
3. m23  = matrix(1:6,2:3)        # m23 = 1 3 5
                                 #       2 4 6
4. m23  = matrix(1:6,2,3,        # m23 = 1 2 3
                 byrow=TRUE)     #       4 5 6
5. v24 = 1:24;                   # 24-element vector.
6. a234 = array(v24,c(2,3,4))    # a 2-by-3-by-4 array.
```

The function `matrix` can only create 2-dimensional arrays: rows, columns, and matrices, whereas the function `array` can create arrays of any dimensions, including matrices. The first argument in either case is an array of data. The second argument to `matrix` may be a vector of two values, the dimensions of the matrix, as in line 3 above. Or the 2nd and 3rd arguments may be scalars, often a more convenient way to give matrix dimensions as in line 4 above. The default arrangement of elements is columnwise, as in line 3, but this can be overridden by setting the argument `byrow=TRUE` as in line 4.

In example 6 above, there are two arguments: a vector of data and a vector of dimensions. If the arguments are arrays, they will be coerced to vectors. When using the function `array`, the dimensions must be defined by a single vector, not a series of scalars.

2.13 Indexing Rectangular Arrays

To index an array, you need as many indexes as there are dimensions. Other than that, indexes work on rows, columns, pages, etc. much as they do on elements of a vector (see above). Examples with matrices should suffice to illustrate:

```
m44 = matrix(1:16,4,4,        # m44 = 1  2  3  4
             byrow=TRUE)      #       5  6  7  8
                              #       9 10 11 12
                              #      13 14 15 16

x = m44[2:3,c(T,F)]           # x =  5  7
                              #      9 11

x = m44[-(2:3),c(T,F,F,T)]    # x =  2  3
                              #     14 15

m44[2:3,c(F,T,T,F)] = 0       # m44 = 1  2  3  4
                              #       5  0  0  8
                              #       9  0  0 12
                              #      13 14 15 16

v = m44[,4]                   # v = 4 8 12 16,  a vector!
m41 = matrix(m44[,4],4,1)     # m41 = 4 8 12 16,
                              #       a column matrix.

dim.m44 = dim(m44)            # Get dimensions (vector) of m44.
dim(m44) = c(2,8)             # Set dimensions to 2-by-8.
```

The examples above show two conventions. First, note that the first index in `m44[,4]` is missing. This means use all rows. If the second index were missing, it would mean use all columns. Second, although `v` has been extracted from a matrix, it has only one row. It is therefore regarded as a vector, and does not retain the row-column orientation of its source. Similarly, when a matrix is extracted from a 3-D array, there are three ways to do this:

```
# i is a scalar, a is a 3-D array.
1. m = a[i,,]   # ith row, all columns and pages.
2. m = a[,i,]   # ith column, all rows and pages.
```

```
3. m = a[,,i]   # ith page, all rows and columns.
# In all 3 cases, m is a 2-D matrix, with no memory
# of which dimension got dropped.
```

In other words, R tries to simplify dimensionality whenever it can. You can override this convention by setting the parameter drop to FALSE as follows:

```
m22 = matrix(1:4,2,2)        # m22 = 1 3
                             #       2 4
v2 = m22[,2,drop=FALSE]      # v2 = 3
                             #      4
a222 = array(1:8,c(2,2,2))  # a222 = a 3-D array.
k22 = a222[,,2,drop=FALSE]  # k22 = 5 7
                             #       6 8
```

The object v2, while having only one column, is a 2-by-1 matrix, not a vector in the R sense, i.e. class(v2) is "matrix" like m22. The object k22, while having only one page, retains its status as an array, i.e. class(k22) is "array" like a222. In general, you can elect to retain the dimensionality of the original array when extracting lower dimensional subsets.

2.14 Matrix and Array Arithmetic

The infix operators listed above also apply to matrices and arrays in element-by-element fashion. The result is an array of the same size as the two operands. However, in contrast to the free-wheeling operations on vectors, strict conformity is the rule. Infix operations apply only to two arrays with the same number, value, and order of dimensions. For example, a row cannot be added to a column.

In linear algebra, there is another kind of multiplication called matrix multiplication, which is not element-for-element:

```
                C = A %*% B
The operation  %*% is infix matrix multiplication.
The number of columns in A and the number of rows in B
must match, call it n.  C will have the number of rows in A
and the number of columns in B, and each element C[i,j]
will be defined as:
C[i,j] = sum-over-k( A[i,k] * B[k,j] ), k = 1 to n.
```

This operation is only defined for matrices, including rows and columns, provided the size requirements are met.

2.15 Vectors Interacting with Rectangular Arrays

Vectors have privileges when interacting with arrays. A vector and an array may be the two operands of an infix operator, e.g.:

```
# A1 is an array, v is a vector.
A2 = A1 / v   # A2 is an array, same dimensions as A1.
```

To visualize this, think of the array A1 as converted to a vector, with its elements arranged column under column. The arithmetic then proceeds as for two vectors, provided v does not have more elements than A1. That is, the elements of v can be repeated as needed to cover every element of A1, but A1 will not be expanded to accommodate v, nor will v be truncated to match A1. If v has more elements than A1, an error occurs.

Vectors have privileges in matrix multiplication as well:

```
 1. M34 = matrix(1,3,4)     # M34 = 3-by-4 matrix of 1's.
 2. v3  = 1:3               # v3  = 1 2 3    (vector)
 3. v4  = 1:4               # v4  = 1 2 3 4  (vector)
 4. r3  = matrix(v3,1,3);   # r3  = 1 2 3    (row)
 5. c4  = matrix(v4,4,1);   # c4  = 1 2 3 4  (column)
 6. x   = r3 %*% M34        # x   = 6 6 6 6  (row)
 7. x   = v3 %*% M34        # x   = 6 6 6 6  (row)
 8. x   = M34 %*% c4        # x   = 10 10 10 (column)
 9. x   = M34 %*% v4        # x   = 10 10 10 (column)
10. x   = r3 %*% t(r3)      # x   = 14,      (scalar)
                            #  (t is the transpose function)
11. x   = r3 %*% v3         # x   = 14,      (scalar)
```

As lines 6-11 above illustrate, in matrix multiplication, a row or column can be replaced by a vector with the same number of elements, as long as there is some reasonable way to determine the vector's implied row-column orientation. Matrix multiplication is strict about dimension. Vectors must have the proper number of elements for the matrix operation in question. In matrix multiplication, vectors will not be expanded to fit the expected sizes of rows or columns.

2.16 Sizes of Vectors, Matrices, and Arrays

To determine the number of elements in a vector or matrix or array, use the function `length` as in lines 6 and 7 below. Recall that vectors have no dimensions in the matrix algebra sense. To determine the dimensions of a matrix or array, use the function `dim` as in lines 8 and 9 below. The function `dim` accepts array arguments, but produces a vector result. Since vectors don't have dimensions. If `v` is a vector, the result of `dim(v)` is `NULL`. However, every object in R has a length.

```
1. v6   = 1:6;                  # v6   = 6-element vector.
2. r6   = matrix(v6,1,6);       # r6   = 6-element row.
3. c6   = matrix(v6,6,1);       # c6   = 6-element column.
4. m23  = matrix(v6,2,3);       # m23  = 2X3 matrix.
5, a234 = array(1:24,c(2,3,4))  # a234 = 2X3X4 array.
6. lv6  = length(v6)            # length(v6) = 6.
7. la234 = length(a234)         # length(a234) = 24.
```

```
8. dimc6 = dim(c6)              # dim(c6) = 6 1 (vector).
9. dimr6 = dim(r6)              # dim(r6) = 1 6 (vector).
```

2.17 Lists

A list is an object whose elements can be other objects: vectors, matrices, arrays, and even other lists. It is called a recursive object, because its elements can contain objects like itself. As a trivial example, we can store a vector of numbers as a list by using the `list` function:

```
Lst = list( staff=6, teachers=35, students=514 )
stff=Lst[[1]];  tchrs=Lst[[2]];  stdnts=Lst[[3]]
stff=Lst$staff; tchrs=Lst$teachers; stdnts=Lst$students
```

Line 1 defines `Lst` to be a list of three numbers associated with the names `staff`, `teachers`, and `students`. We can now access these numbers either by name or by index, e.g. lines 2 and 3 above are equivalent. Note that elements of a list are indexed by double brackets as in `Lst[[1]]`.

A richer example is given below, where a different kind of data is stored in each element of the list:

```
Lst = vector( "list", 5 )  # Lst is a list of 5 NULL elements.
Lst[[1]] = 6                 # scalar
Lst[[2]] = 1:6                 # 6-vector
Lst[[3]] = matrix(1:6,2,3)       # 2x3 matrix
Lst[[4]] = array(1:24,c(2,3,4))    # 2x3x4 array
Lst[[5]] = list(1,2,3,5,8)       # list of 5 scalars.
```

`Lst` now contains five distinct kinds of numerical data. In order to access a number within `Lst`, we must first specify the element of `Lst`, and then the particular number within that element, e.g. continuing the code from above:

```
# Statement         # Result in x
x=Lst[[1]]          # 6,  from scalar
x=Lst[[2]][3]       # 3,  from 6-vector
x=Lst[[3]][1,3]     # 5,  from 2x3 matrix
x=Lst[[4]][1,2,3]   # 15, from 2x3x4 array
x=Lst[[5]][[5]]     # 8,  from list of 5 scalars.
```

The index of the element of `Lst` is specified in double brackets, while indexes of vectors and arrays within those elements are in single brackets.

Chapter 3

Strings

3.1 Forming Strings

A string is formed by enclosing characters, including blanks, between matched pairs of double quotes:

```
str1 = "This is a string."
```

Apostrophes, as in `isn't`, or pairs of single quotes as in `"'Why?' he asked."` are permissible in strings.

A string of one or more characters is a data item of mode `character`, just as a real number is a data item of mode `numeric`. Using an assignment statement, a string can be stored in a scalar, or in an element of a vector or array. In determining the mode of a vector or array, strings take precedence over numbers, so that a string stored in a single element of a large numeric array will convert the entire array to mode `character`. That is, all the existing numeric elements will be converted to strings: e.g. the value `1.5` will be converted to the string `"1.5"`

3.2 Putting Strings Together

Several strings can be combined into one using the `paste` function, e.g:

```
s1="Yankees,";  s2="go";  s3="home!"
s4 = paste(s1,s2,s3)          # s4 gets "Yankees, go home!"
s5 = paste(s2,s3,sep="")      # s5 gets "gohome!"
s6 = paste(s2,s3,sep=" on ")  # s6 gets "go on home!"
```

By default, `paste()` puts blanks between its arguments, as in `s4` above. To change the default separator, the argument `sep` may be set to the null string as in `s5` above, or to some preferred value as in `s6` above. See more about `paste` below.

3.3 Accessing Parts of Strings

Indexes are used to access elements of vectors and arrays. Since a string is a single element, indexes refer to entire strings, e.g. in a string vector, not to parts of a string itself. Parts of a string can be accessed using the `substr` function:

```
1. s1="What is the time?."
2. s2=substr(s1,6,11);       # s2="is the"
3. substr(s1,1,7)="Is this"  # s1="Is this the time?"
```

The first argument to `substr` is the name of the string, while the second and third arguments are the indexes of the first and last character of the substring we wish to get or put. Thus in line 2 above, the string `"is the"` is the 6th through 11th characters in `s1`. `substr` can be used to extract a substring as in line 2, or to change a substring as in line 3.

3.4 String Vectors and Arrays

Strings, like numbers, can be stored in vectors and arrays. The indexing conventions for storage and retrieval are the same in either case, even though strings can be of arbitrary and non-uniform length. A string vector can be created using the `c` function:

```
# charvec1 is a vector of mode "character" with 4 elements:
charvec1 = c("z", "1945", "Short String",
   "A rather long, ornate, and verbose string")
# charmat is a 2x2 matrix:
charmat = matrix(charvec, 2, 2)
# charvec2 is a vector of 8 elements:
charvec2 = c("a","b","c","d","e","f","g","h")
# chararray is a 2x2x2 array:
chararray = array(charvec2,c(2,2,2))
```

3.4.1 String vectors and paste

As with numeric vectors, string vectors have special privileges. For example, the function `paste` coerces its arguments to string vectors. It then pastes strings together element-for-element.

```
sv1 = c{"a","b","c","d");
sv2 = c("x1","y2");
sv3 = paste(sv1,sv2,sep="")
# sv3 gets c("ax1","by2","cx1","dy2").
sv4 = paste(sv2,sv1,sep="",collapse="");
# sv4 gets the single string "x1ay2bx1cy2d"
```

If some vectors are shorter than others, `paste` recycles the shorter vector as numeric vectors would be recycled. Notice in the above example how `sv2`, which is shorter than `sv1`, is recycled in `sv3`. The parameter `sep` is the separation character inserted when the strings are pasted together. In `sv3`, `sep` is set to the empty character, i.e. no character and no space. When the parameter `collapse` is specified (not NULL), paste assembles the resulting vector into a single string, inserting the string `collapse` between the elements.

3.5 Coercion

It is often necessary to change one type of data to another, e.g. numbers to strings for printing, strings to numbers for computing, and vectors to matrices for linear algebra. The process of changing one type of data to another, or forcing one data type to act as another, is called coercion. The user can control such events with a set of functions that begin with `as.` For example, if we have a string vector `strnums` in which every string represents a number, we can increase every number in `strnums` by 1, and store the result as a string vector as follows:

```
strnums = c("1.25","-7.75","1e3")  # Strings
# Convert strings in strnums to numbers, then add:
   sums = 1 + as.numeric(strnums)
# Convert the sums back to character.
   strnums = as.character(sums)
# The same procedure as above in 1 statement by
# coercing strnums to numeric and converting back.
   strnums = as.character(1+as.numeric(strnumms))
# In either case, the result should be:
# strnums = c("2.25","-6.75","1001")
```

We must coerce `strnums` to numeric in order to add 1, because `R` does not permit arithmetic operations on strings. As another example of coercion, whenever an array is used as an index, it is implicitly coerced to a vector. Here is a list of `R` coercion functions:

```
as.array  as.character  as.complex  as.data.frame
as.data.frame.table  as.difftime  as.double  as.factor
as.is  as.list  as.logical  as.matrix
as.numeric  as.raw  as.ts  as.vector
```

Exploring these functions with the help files is one way to get introduced to the kinds of objects `R` works with.

3.6 String Functions

R has a rich assortment of functions to assist in manipulating strings and string vectors. See the “Function Survey” chapter under “String Functions”. For further details on these functions, see the help files and examples.

Chapter 4

Mixing Numbers and Strings

4.1 Names of Elements, Rows, Columns, et al.

Vectors and arrays can have names associated with their parts. Names are often easier to remember than indexes. Of course, many languages can label rows and columns with names when printing, but `R` goes further in that the names become an integral part of the object:

```
# A vector of prime numbers <20:
   primes = c(2,3,5,7,11,13,17,19);
# Name these with their indexes, and print them:
   names(primes) = 1:length(primes);  primes
        1   2   3   4   5   6   7   8
        2   3   5   7  11  13  17  19
```

It may seem that `names(primes)` is a vector of numbers, but in fact they have been coerced to strings and printed without quotes. We can redefine the names, then print as follows:

```
names(primes)=c{"1st","2nd","3rd","4th",
                "5th","6th","7th","8th")
primes
     1st 2nd 3rd 4th 5th 6th 7th 8th
       2   3   5   7  11  13  17  19
```

We can now access the fourth prime (7) either by its numeric index: `prime[4]`, or by its string name: `prime["4th"]`.

4.2 Matrix Row and Column Names

A matrix may have names associated with its rows or its columns or both:

```
# M33 is a matrix of prices of 3 items (rows)
#    at 3 stores (columns).
M33 = matrix(c(1.99,1.89,2.19
               2.49,2.19,2.75,
                 4.29,4.29,4.59),3,3,byrow=TRUE)
rownames(M33) = c("Bread","Eggs","Detergent")
colnames(M33) = c("Trader.Ted's","Cheap.Gourmet","McDee's")
M33
          Trader.Ted's Cheap.Gourmet McDee's
Bread             1.99          1.89    2.19
Eggs              2.49          2.19    2.79
Detergent         4.29          4.29    4.59
M33["Eggs","Cheap.Gourmet"]   # or M33[2,2]
[1]  2.19
```

As with names of vector elements, row names and column names can be used as indexes.

4.3 Array Dimension Names

Just as a matrix can have names for one or both dimensions, an array can have names for any dimensions one chooses: In the following example, we have set up a 3-by-4-by-6 array, called `A346`, of prices for 3 items in 4 store chains in 6 cities. Now we wish to denote which items, store chains, and cities are being priced:

```
dimnames(A346) = list(   # a list of string vectors.
   # Row names first:
   c("Bread","Detergent","Eggs"),
   # Column names second:
   c("Cheap.Gourmet","McDee's","Trader.Ted's","WallMark"),
   # Page names third:
   c("Miami","Atlanta","Washington",
     "Philadelphia","New York","Boston"))
```

The appearance of `A346` on printout is left as an exercise. We do not have to assign names to all dimensions. In the above example, the line beginning `c("Trader` can be changed to `NULL`. As a result, the columns of `A346` will be labeled by number instead of store names.

4.4 Named Lists

When a list is defined, names may be given to the elements, but the elements themselves may be any type of data, including other lists, arrays with named dimensions, and so on:

```
Lst = list( Name1=Data1, Name2=Data22 )
```

There are two ways to access the elements of a list:

```
by index,   e.g.:  Lst[[i]]  (note double brackets)
or by name, e.g.:  Lst$Name1
```

If the elements in of a list are also lists, they can be accessed in the following fashion:

```
# Create 2 lists, Lc1n1 & Lc2n2, each containing
#  a string vector and an integer vector:
     Lc1n1 = list(C1=c("a","b"), N1=c(1,2))
     Lc2n2 = list(C2=c("c","d"), N2=c(3,4))
# Make a list containing copies of Lc1n1 & Lc2n2
#  renamed LA & LB:
     L3 = list(LA=Lc1n1, LB=Lc2n2)
# Remove Lc1n1 & Lc2n2, which are now
#  redundant because L3 has copies LA & LB.
     rm(Lc1n1,Lc2n2)
# There are 4 ways to get at e.g. the element "d",
#  which is the 2nd element in vector C2,
#  which is the 1st vector in list LB,
#  which is the 2nd list within list L3:
     x = L3  [[2]] [[1]] [2]   # x = "d"
     x = L3  $LB   [[1]] [2]   # x = "d"
     x = L3  $LB   $C2   [2]   # x = "d"
     x = L3  [[2]] $C2   [2]   # x = "d"
```

At each level of named access, either the index (enclosed in double brackets) or the name (preceded by $) may be used.

4.5 Workspaces (Environments)

Every object resides in some workspace. In a typical `R` session, workspaces are created and discarded frequently. Your variables in your master session reside in a workspace called `".GlobalEnv"`, which is typically the first of several workspaces (i.e. level 1), and is therefore the first place `R` looks to find objects. Subsequent workspaces are packages of functions for statistics, graphics, utilities, etc. The final package is `base`, which contains over a thousand functions, incuding basic math and those functions that create, coerce, and test for those objects we have discussed in these notes. If you create a function named e.g. `cos`, that function resides in your level 1 workspace, and it has precedence over the `cos` that resides in `base`.

Every time a function is called, its variables reside in the function's private workspace. When a function finishes its task, its workspace is discarded. However, if a function `F1` calls another function `F2`, then `F1`'s workspace is pushed down in the stack of function workspaces while `F2`'s workspace takes over. Even if a function calls itself (such recursive calls are permitted), a new workspace is created for each level of call.

To show current workspace names and their indexed order, say `search()`. To show the contents of any workspace, say `ls(n)` where `n` is the index of the workspace, or say e.g. `ls(".GlobalEnv")`.

4.6 attach and detach

To avoid the complexity of indexing a list, `R` provides a means to get at its elements without the use of dollar signs or multi-leveled indexes. The functions `attach` and `detach` serve this purpose. In the above example, we have removed `L1` and `L2`. If we now say `attach(L3)`, the lists `L1` and `L2` from within `L3` become directly accessible as though they were sepapate variables.

Every call to `attach` creates a new workspace at level 2, just below `".GlobalEnv"`. In this case, the workspace will be named `"L3"`. So if you already have variables named `L1` and `L2` in your workspace (level 1), they will mask those you just created in level 2. You will be warned about this, and if those level 1 values are no longer needed, you can say `rm(L1,L2)`. The level 1 values will be removed, so that you can use the new level 2 values. However, if you now alter e.g. `L1`, another object named `L1` is created at level 1, again masking the level 2 object. In effect, attached objects are read-only.

To make the example even more complex, if we then say `attach(L2)`, `L2`'s elements `C2` and `N2` likewise become directly accessible in a new level 2 workspace called `L2`. Beacuse two workspaces cannot ocupy the same level, workspace `"L3"` is pushed down to level 3. If `"L2"` (at level 2) had variables with the same names as `"L3"` (at level 3), `R` would warn of this, and the duplicate names in `"L2"` would mask those in `"L3"`.

We can reverse the process partially with `detach(L3)`, which removes workspace `"L3"` but that leaves `C2` and `N2` still available in wokspace `"L2"`. We can complete the restoration by saying `detach(L2)`. In such multiple attachments, it seems intuitive to attach as necessary, then detach in reverse order, but it isn't required.

A call to `detach` removes variables and workspaces, but any altered versions of those variables have been added to level 1 and remain there. To alter the original list, you must store the changes directly, e.g. `L3$L1=L1`.

For further peculiarities, see the chapter: "Program Modules" in the subsection: "Attached Lists within Functions".

4.7 Factors

At first glance, a factor looks like a string vector, but it is in fact an integer vector disguised by a name for each level. Its purpose is to provide a user-friendly format (names) for potentially cryptic data (integers). Factors are formed by the function `factor`, or by coercing string vectors using `as.factor`. For example, in a certain taste test, the permissible responses are:

```
"hate"  "dislike"  "tolerate"  "like"  "love"
```

which means there will be five levels. Suppose 50 people take part in the test. We store their responses, chosen from the above list, in a string vector called `Cresp`, but string vectors are awkward for polling purposes. We convert `Cresp` to a factor `Fresp`:

```
# Cresp is string vector of 50 responses.
# L is a list of permissible responses (levels),
#   in the desired order.
```

```
   > L = c("hate","dislike","tolerate","like","love")
# ordered=TRUE means the levels will be numbered 1 to 5
#    (implicitly) in the order given
#    by the "levels" parameter:
   > Fresp = factor(x=Cresp, levels=L, ordered=TRUE)
   > Fresp   # Print Fresp (next 2 lines)
   [ ]  (THE 50 RESPONSES IN THEIR ORIGINAL ORDER
   [ ]   ARE PRINTED ON THESE LINES.)
   levels: hate < dislike < tolerate < like < love
# The above line with < signs shows that
#    the levels have a deliberate
#    (rather than default) ordering.
   > summary(Fresp) # Print a summary of results.
     hate  dislike  tolerate  like  love
        6        9        20    12     3
```

A factor can have more levels in the ”levels” vector than are represented in the actual data. For example, if there were no ”tolerate” votes in `Fresp`, the above summary would list zero instead of 20. Unrepresented levels may become represented when new data is introduced, e.g. when dataframes are merged.

This simple summary is by no means the limit of a factor's uses. In the context of dataframes (discussed below), factors are so useful that any string vector stored in a dataframe is coerced to a factor by default.

4.8 Dataframes

A dataframe is a specialized named list. Each element of a dataframe is a vector that may contain a class of data distinct from the other vectors. That class is consistent throughout the vector, e.g. numbers, strings, factors, dates, or logicals. All vector elements of a dataframe have the same length, so that the appearance when displayed (with the vectors as columns) is that of a spreadsheet.

Dataframes are usually read from files, but they can also be created from scratch in a manner similar to creating named lists:

```
Cvec = c( "ab", "cd", "ef" )
Nvec = c(  12,   34,   56  )
# Create and display the dataframe df.
df = data.frame( C=Cvec, N=Nvec );  df
   C  N
1 ab 12
2 cd 34
3 ef 56
# A dataframe may be formed from an existing list
df = as.data.frame(ListA)  # ListA is coerced to dataframe.
Q = is.data.frame(df)  # Q is TRUE if df is a dataframe.
```

However, these are toy examples, and do not begin to exhibit the richness of dataframes. A separate chapter of these notes is devoted to dataframes.

Chapter 5

Program Flow

5.1 Expressions

One form of statement in R is an expression e.g. `sin(2*pi*x)` which produces a value. If you type an expression alone on a line, its value will be printed and then discarded. An expression is an object in R. It can be saved, evaluated ,and manipulated:

```
ex1 = expression( x^2 + 2*x + 5 );     # Stored expression
exv = expression( x+1; 2*x+2, 3*x+3 )  # Vector of expressions
x=5
y = eval(ex1)                          # y=40, ex1 evaluated.
dex1 = D(exi,x)                        # dex1 = 2*x+2, the
                                       #   derivative of ex1.
```

5.2 Assignments

To save the value of an expression, an assignment e.g. `y=sin(2*pi*x)` is used, which stores the value of the expression to the right of = in the target variable to the left (y in this case). There are two other assignment symbols, <- and -> (2 characters each) as in:

```
y <- sin(2*pi*x)   # Target variable on the left.
sin(2*pi*x) -> y   # Target variable on the right.
```

For typing convenience and visual simplicity, I prefer =, since (to me) `x<-6` looks like x is less than minus 6.

5.3 Groups

Sometimes, it is desirable to treat a group of statements as a single entity. This is done by enclosing the group of statements in braces as in:

```
{x2=x*x; x3=x2*x; x4=x3*x}
```

The group enclosed in braces can be a single expression, a single assignment, or a series of expressions, assignments, and other groups. Just as expressions and assignments produce single values (which may be compound objects like lists), so a group also produces a single value, namely the value of the last expression evaluated by the group. In other words, while the group may carry out many assignments, the group as a whole may be used as an expression as in:

```
y = sin(x) + { t=tan(x); z=cos(x) }
# t   gets   tan(x),   and
# z   gets   cos(x),   while
# y   gets   sin(x) + cos(x)
# Note that tan(x) is not added to y
# because it is not the last expression
# in the group { t=tan(x); z=cos(x) }
```

This convention also allows us to assign values to many targets in one statement:

```
x1 = 2;               # x1 = x^1
x5 =      x1 * {      # x5 = x^5
     x4 = x1 * {      # x4 = x^4
     x3 = x1 * {      # x3 = x^3
     x2 = x1*x1}}}    # x2 = x^2
```

In the above example, evaluation of `x5` begins with the innermost statement (the last line), and works upward, so that values are assigned to `x2, x3, x4,` and `x5` in that order.

In the discussion that follows, let `Lgc1, Lgc2`, etc. be logical expressions, and let `St1, St2`, etc. be single statements (not enclosed in braces) or statement groups (enclosed in braces). Also let `Cd1, Cd2`, etc. be bodies of code

5.4 Statements that Choose

5.4.1 if: A 2-Way Branch

When a choice must be made between two alternative actions, i.e. a two-way branch, an `if` construct is used:

```
if(Lgc1) St1    # If Lgc1=TRUE, do St1.
else     St2    # 'else St2' may be omitted,
                #    but if present and
                #    Lgc1=FALSE, do St2.
                # After St1 or St2,
St3             #    control passes to St3.
#  if, used as a function, returns a value, e.g.:
x = if(lgc2) 3 else 5
# if Lgc2 is TRUE, x gets 3, else x gets 5.
```

5.4.2 ifelse: a 2-Way Function

Just as `if` can be used as a function to choose between a pair of alternatives, so the `ifelse` function can choose between many pairs of alternatives in array fashion. For example, suppose we wish to generate a vector of of values `v` from 3 given vectors `a, x,` and `y`:

```
# Let a, x, and y be vectors of equal length.
# Let med = median(a).
# If a[i]<med, v[i] gets x[i], but
#    if a[i]>=med, v[i] gets y[i].
# This can be done using if in a loop:
for (i in 1:length(a))
   v[i] = if (a[i]<med) x[i] else y[i]
# However, ifelse does it on entire arrays without a loop.
v = ifelse( a<med, x, y)   # v is same size as a.
# Alternately, the elements of array a
#   can be labeled "small" or "large":
alabel = ifelse( a<med, "small", "large")
# alabel is same size as a.
# alabel[i] gets "small" or "large"
#    depending on the size of a[i].
```

The function `ifelse` can operate on vectors and arrays. In the above example, if vectors `x` and `y` are shorter than array `a`, the usual expansion rules for vectors apply. Likewise, the above strings `"small"` and `"large"` are scalars that must undergo vector expansions if `a` has 2 or more elements.

5.4.3 switch: An n-Way Branch

The function `switch` can chose among n alternative expressions or assignments:

```
# x = a vector of data.
# n = an integer in 1:3.
str = switch( n, "Mean", "Median", "Geom",    # Example 1
                 stop("Unknown measure") )
m = switch( str,                              # Example 2
            Mean = mean(x),
            Median = median(x),
            Geom = exp(mean(log(x)))  )
dummy = switch( n,                            # Example 3
     {St1},{St2},{St3},...,{StN},
     stop("Error: n is not in 1:N")  )
```

Note that `switch` is a function, not a statement. See more about functions below, but here we will point out a few essentials. A function is invoked by its name, followed by arguments enclosed in parentheses. The first argument to `switch` is always the selector (a number or string) that is used to choose one of the several alternative expressions, statements, or groups that follow.

In Example 1 above, the string variable `str` will receive one of the three given strings depending on the value of `n`, e.g. if `n=3`, then `str="Geom"`. If `n` is not in `1:3`, the stop clause is used, giving an error message. The `stop` clause is optional. If `stop` is omitted and `n` is out of range, `switch` returns a NULL value.

In Example 2, the numeric variable `m` is to receive a measure of vector `x`. The particular measure is chosen by matching `str` with one of the three variable names (targets) below it. Only the statement whose target name matches the string in `str` will be executed, e.g. if `str="Geom"` then the statement with the target `Geom` will be used, but not `Mean` or `Median`.

In example 3, `n` directs the program flow to one of several statement groups `St1` through `StN`, with an error stop if `n` is greater than the number of groups. Thus `switch` is a general N-way branch. For an application of this, see the section on spaghetti code below.

5.5 Branching by Index

A script `scr1` is invoked by `source("scr1.R")`. The argument to `source` is a string that could be e.g. from a vector. Thus the index that chooses the script name is effectively a program branch:

```
# scrvec is a vector of script names:
  scrvec = c( "scr1.R", "scr2.R", "scr3.R" )
# Branch to the ith script named in scrvec:
  source(scrvec[i])
```

Recall that functions and expressions are objects in R. Vectors and lists of such objects can be constructed. Then branches can be chosen based on indexes:

```
# exvec is a vector of expressions:
   exvec = expression( 3*x+2, 8*x+5, 13*x+8 )
# y gets the value of the ith expression in exvec:
   y = eval(exvec[i])
# TrigList is a list of trig functions:
   TrigList = list( cos, sin, tan )
# trigfunc becomes the ith function in TrigList.
   trigfunc = TrigList[[i]]
   y = trigfunc(pi/4) # y gets a value of that function.
# y gets a value of the ith function
#    directly from TrigList:
   y = TrigList[[i]](pi/4)
```

5.6 for Loops, next, and break

When a statement group is to be repeated a specified number of times, the `for` loop is used. The basic form of the `for` loop is:

```
          for(x in W) st1     # See item 1 below.
```

Most of the time, `St1` will be a group of statements, and often, the complete loop will not be executed because of `next` or `break` statements:

```
          for(x in W){        # See item 1 below.
             Cd1              # Cd1 is executed each time.
             if(Lgc1) next    # See item 2 below.
             Cd2              #
             if(Lgc2) break   # See item 3 below.
             Cd3
          }
          Cd4
```

1. `W` may be a vector or a list. Most of the time, `W` will be a simple vector of numbers like `1:n`. At the start of the loop, a copy of `W` is made, call it `Wcopy`, which will be discarded after the loop. `Wcopy` cannot be changed by the user, even by altering W during the loop. The `for` loop executes as many times as there are elements in `Wcopy`. If `Wcopy` is a matrix or array, it is coerced to a vector. Then for each successive loop, `x` gets the value of the next element of `Wcopy`. When the elements of `Wcopy` are used up, control is passed to the statement after the loop. (Cd4 in the example).

2. The `next` command skips the remainder of the current loop and goes on to the next, i.e. `next` returns to the top of the loop for the next iteration with the next value of `x`, if any. In the example, if `Lgc1` is true in this iteration, then Cd2 on down are skipped, and control passes back to the `for` statement.

3. The `break` command terminates the loop, and passes control to the statement after the loop. (`break` passes control directly to Cd4 in the example). If a `break` statement is within nested loops of any kind, (e.g. a `for` loop within a `while` loop, discussed below) it breaks out of the innermost loop only.

4. The `for` loop is the safest loop in R, because it is guaranteed to end when the elements of `Wcopy` are used up. The next two loops have no such safeguards.

5.7 while and repeat Loops

When the number of iterations of a loop is uncertain, a `while` loop or `repeat` loop may be required. The form of a `while` loop is:

```
               while(Lgc1)  Cd1
```

The code `Cd1` is repeated until the logical expression `Lgc1` becomes `FALSE`, or a `break` statement is encountered. However, unlike the `for` loop, R cannot guarantee that the `while` loop will ever terminate.

Sometimes it is difficult or inefficient to express the loop termination condition as a single logical expression, in which case, the `repeat` loop is used:

```
                repeat   Cd1
```

Somewhere in code `Cd1`, there must be a `break` statement (and some means of getting there!), or the `repeat` loop will run forever.

The statements `next` and `break` do the same things in `while` and `repeat` loops as they do in `for` loops.

5.8 Spaghetti Code

R has no "go to" statement, because of the legendary confusions associated with arbitrary leaps from one part of a code to another. A plate of spaghetti is an apt metaphor for the flow of such programs. The grammar of R is intended to encourage "top-down" programming that runs from some start (the top) to some finish. However, there are methods in the literature that simply are not written this way. Suppose you are faced with programming the following nightmarish recipe (most of which is hidden in the Blocks):

Step 0: Initialize variables.
Step 1: Do Block 1. Go to Step K.
Step 2: Do Block 2. Go to Step K.
Step 3: Do Block 3. Go to Step K.
Step 4: End.

K is set to 1, 2, 3, or 4 by each Block according to the recipe. The above program may bounce around chaotically between Blocks 1, 2, and 3, with no assurance that Step 4 will ever be reached. Regardless, we are obliged to programm this procedure in R, which has no "Go to" statement. We can proceed as follows. First we write scripts called `Block0.R`, `Block1.R`, `Block2.R`, and `Block3.R` from the published recipe. (See more about scripts below.) Then we set up a `repeat` loop with an embedded `switch` that chooses the next Block based on K:

```
source("Block0.R"); K=1   # Initialize variables.
repeat                    # repeat loop.
   switch(K,              # switch according to K.
      source("Block1.R")  # Do this when K=1.
      source("Block2.R")  # Do this when K=2.
      source("Block3.R")  # Do this when K=3.
      break)              # Stop when K=4.
```

The moral of this example is, "You can write spaghetti code in any language."

Chapter 6

Program Modules

6.1 Scripts

In an R session, you give commands. When a given series of commands is to be used often, rather than type it in repeatedly, it is more convenient to put it in a script that is stored in a file (e.g. `myscript.R` below) which can then be invoked in a single statement:

```
y=source("mysrcipt.R")   # "y=" is optional.
```

A script contains commands that are executed exactly as though you had typed them in. The variables used by the script are the variables in your workspace. If a script creates variables, they appear in your workspace. A script can contain all the grammatical constructs like `if` and `for` and calls to other scripts and functions.

The `source` function that invokes scripts also returns the value (y in the above example) of the last expression evaluated by the script.

6.2 Functions

A function's usual role is to accept arguments and produce a single result. That "single" result may be any of the objects from the past chapters, e.g. a vector or list, so it is not a significant restriction. The value returned by the function is the last expression it evaluates. Some functions are quite short, and they can be defined "on the fly" during a session, e.g.:

```
myfunc = function(x) a*x^2+b*x+c
```

Such functions can be defined directly by you or in scripts. Once defined, functions can be invoked at any time by e.g.:

```
z = myfunc(x)*myfunc(y)
```

6.3 Function Calls

The syntax for writing a function is:

```
func = function(arg1,arg2,arg3,arg4,etc.) Cd1
# etc. here means continue as needed.
```

where `func` is the name of the function, `arg1`, etc., are the input arguments, and `Cd1` is the code of the function, which may be an expression, an assignment, or a group enclosed in braces. Some syntaxes for invoking the above function are:

```
z = func(Ar1,Ar2,Ar3,Ar4)
z = func(Ab1,Ab2) + func(Br1,Br2,Br3)
```

Functions can be written to accept varying numbers of arguments. Documentation should make it clear how many arguments are required, and what happens when some are left out. In line 1 of the above example, the user's variable `Ar1` will be used as the function's argument `arg1`, and so on in order of appearance. The user's variable name and the function's argument name need not match as long as the order is observed. However, e.g. if you wish call `func` giving `arg1`, `arg3`, and `arg4`m while omitting `arg2`, the actual argument names from the code of `func` must be used:

```
z = func( Ar1, arg4=6, arg3="x-y plot")
```

Arguments `arg3` and `arg4` are in reverse order. When the argument values are associated with explicit argument names, the order becomes irrelevant. Also, the function `func` is given no value for `arg2`. In this case, `func` must be coded to provide a default value for `arg2`, e.g.:

```
func = function( arg1, arg2=0, arg3, arg4 )
# When arg2 is omitted, its value will be 0.
```

6.4 Functions Everywhere

Every action requested by an `R` expression (add, compare, save, et al.) is done by a function. An understanding of this principal can be useful, especially when seeking help on various features of `R`. Of course, we can guess (correctly) that computing `sin(x)` is accomplished by invoking a function called `sin`. But some common operations are not so obvious. for example, the expression `a+b` invokes a function called `‘+‘`, *including the tick marks*, so that `a+b` is actually parsed as `‘+‘(a,b)`. (You can write every `R` operation in this manner if you choose, but no one does.) Let's see what happens in a non-trivial example:

```
# The statement as we type it:
    v = r*(s+t/u)
# The statement as R parses it:
    ‘<-‘( v, ‘*‘( r, ‘+‘( s, ‘/‘(t,u) ) ) )
# Note, R’s name for = (i.e. assign) is ‘<-‘.
```

Once we know the convention about putting tick marks around operations, we can guess the function names of all the arithmetic operations. (Double quotes work too.) But the various indexing operations are a bit harder. When indexing a source as in `x=v[2]`, the indexing step is done by the function `‘[‘`, but when indexing a target as in `v[2]=x`, the indexed store is done by the function `‘[<-‘`. For a more complete list of such functions, see the “Function Survey” chapter in the section “Assignment and Indexing Functions”.

6.5 Unnamed Functions

As we saw in the previous section, a function can be invoked by name followed by an argument list enclosed in parentheses. It is also possible to specify a function without naming it. `R` provides functions with the stem `apply`, whose job it is to apply other functions to various kinds of data, e.g.:

Function	Type of data
sapply	vector
apply	matrix, array
tapply	table
lapply	list

For example, `sapply` evaluates functions on vector data, and may be used as follows:

```
# Create a vector x = 0 to 1 by 0.01
   x = seq( 0, 1, len=101 )
# On the vector x, apply the unnamed function
#    defined as:   sin(2*pi*x) + sin(6*pi*x)
   y = sapply( x, function(x) sin(2*pi*x)+sin(6*pi*x) )
# Of course, sapply will apply functions by name, too:
   y = sapply( x, sin )
```

6.6 Functions as Objects

```
# Create a new function called poly:
   poly(x) = function(x) x^2 + x^3
1. y = poly(3)     # y = 36, the value of poly(3)
2. y = poly()      # Error: too  few arguments.
3. y = poly(3,5)   # Error: too many arguments.
4. y = poly("Bob") # Error: wrong data type.
5. y = poly        # Function y(x) behaves like poly(x).
6. poly(2)         # Print 12, the value of poly(x).
7. poly            # Print the function definition.
```

A function name followed by parentheses enclosing a (possibly empty) list of arguments causes the function to be invoked (lines 1-4 and 6 above). If the arguments are too few (line 2), too many (line 3), or of the wrong type (line 4), an error occurs. However, if the function name is used without parentheses (lines 5 and 7), it becomes an object that can be assigned (line 5), printed (line 7), indexed (see previous chapter on "Branching by Index"), or passed to another function as an argument.

6.7 Scopes of Variables

There are three categories of variable within a function: formal, local, and free.

6.7.1 Formal Variables

The names of formal variables appear in the function's argument list (as opposed to the caller's argument list). The values of formal variables depend on what the caller has provided. When the caller omits an argument, a default value may be provided in the function's argument list, e.g.:

```
func = function(arg1, arg2, arg3=NaN){
   BODY OF func GOES HERE}
# CALLING STATEMENTS:
z1 = func(2,5,-1) # All args. are user-defined.
z2 = func(7,18)   # NaN is used for omitted arg3.
z3 = func(7)      # Error: no value for arg2.
```

If an argument is omitted, and the function provides no default, an error occurs.

6.7.2 Local Variables

Local variables are defined within the function by explicit assignment. Even though a local variable may have the same name as one of the caller's variables, its value has no effect on the callers variable or on other variables of the same name anywhere in a nest of calls. The status of a formal variable changes to local whenever the formal variable is redefined or altered in any way. For example, if an argument is a large array, and the function changes one number in it, the function makes a complete and local copy for its own use. This convention protects the caller from unexpected changes in the values of his arguments.

6.7.3 Free Variables

Free variables are those that are used by the function with no initialization therein. The values assumed by free variables depend on the context in which the function is written and used. For example, functions may be nested within other functions, so that a free variable of an inner function can get its value from the functions in which it is nested, or from the caller. If the function call is not nested, it may get its value from the caller. Either way, the name of the variable in the function must match the name of some source in its calling history. If such a variable is not found, an error occurs.

6.7.4 Special Assignments

As with formal variables, if a free variable is altered within a function, its status becomes local, thus protecting values outside the function. There are way to defeat this protection, as with the special assignment `<<-` or with the `assign()` function, e.g.:

```
# Function q contains a special assignment
#   for the target abc.
q = function(x){ y=x+abc; abc<<-abc-1; y}
abc = 20;     # abc is initialized at 20.
z=q(3);       # Now z=23 and abc=19.
```

Special assignment targets are resolved the same way that free variables are resolved, by searching back through the calling history to find a variable of that name. Only this time the target, which is *outside* the local workspace, is altered, thus violating the protection of external values. Further, if no target is found with the right name, special assignment creates the target in the user's workspace.

Because of their potentially far-ranging search for resolution, free variables and special assignments are risky, and should be used sparingly. Occasionally, their use makes sense, as e.g. when a function argument is a large array that will be returned with only minor revision:

```
# M is a really big matrix.
#   to which minor revisions will be made.
newM = function( M, ivec, jvec ) {
   # Code for M.new goes here, containing
   #   a few statements like   M[i,j] = q;
   #   and ending with   M }
```

The revision of the few elements `M[i,j]` will cause the entire matrix to be copied, consuming a lot of time, especially if the function `newM` is used repeatedly. A more efficient code for this task is:

```
AlterMij = function( ivec, jvec )
   # Code for AlterMij goes here,  with
   #   a few statements like   M[i,j]<<-q
   #   and ending with   0 }
```

No copy of `M` is made, and a few elements of the caller's `M` are altered without much overhead.

6.7.5 Attached Lists within Functions

As a rule, when a function returns to the caller, the function's local variables disappear. One exception to this rule is a local list that has been attached within a function. As with any attachment, the elements of the list become available as separate variables at level 2. However, when the function returns, and the list itself apparently disappears, those variables remain accessible to the caller. Oddly, even though the list seems gone, the caller can still detach it!

```
# ----- This is a function called Latch -----
Latch = function(x) {  Cd1  # Latch begins.
   L = list(L1=1,L2="a")   # L is a local list.
   attach(L)               # L yields L1 & L2.
   Cd2 }  # Latch ends but L1 & L2 remain at level 2.
# ----- This is the caller's program -----
z1 = Latch(z2)   # Latch is called.
L1   # L1 is printed without protest.
L    # ERROR! L is unknown.
detach(L) $ Although "unknown", L can still be detached.
```

6.8 Functions Can Call Scripts

Functions can call scripts, but a script takes its values from the user's workspace, not from the function's workspace. So unless the function happens to have free variables whose values come from the user's workspace, and the script alters those values, the function will have no idea what the script has done. Exceptions are global side-effects like creating graphs or altering files.

Chapter 7

Function Survey

7.1 Functions Are Everywhere

Every action requested by `R` code is done by a function. There are thousands of `R` functions, over a thousand in the base package alone. This brief survey alerts the reader to some important `R` functions, particularly families of functions, which can then be explored in the help files. NOTE: when searching the help files for function names that use non-alphameric characters (periods being the standard separator are OK), enclose the name in quotes, as in `?"[<-"` for single-bracket indexed assignments like `x[2]=7`. For the names of functions that do such assignments, see the ”Assignment Functions” section.

7.2 Get or Set

Several function names can serve two roles: first, to get information about an object, and second, to (re)set that information. For example, the function `dim` can get array dimensions or set them

```
DimA = dim(A)       # Get vector of dimensions of A
dim(A) = c(4,2,3)   # Reset dimensions of A.
```

depending on which side of the assignment the function appears. When a function name can do this, the phrase “get or set” will appear in its description. NOTE: the two roles are performed by two functions with two distinct names, e.g. `dim` to get, and `"dim<-"` to set.

7.3 Management

Function	Purpose
attach	Treat parts of lists as separate
citation	Show how to cite R in articles
demo	Give demo
detach	Reverse the effects of attach

`example`	Give examples of function use
`find`	What package is the topic in?
`fix`	Create a spreadsheet from a data frame
`help or ?`	Help on specific functions
`help.search or ??`	Search help files for topic
`install.packages`	Install packages
`library`	Info on, and load, packages
`ls`	Show contents of packages
`names`	Get or set names in lists.
`objects`	List defined objects
`require`	Add required packages
`rm`	Remove objects
`scan`	Versatile input function
`search`	Show attached databases
`summary`	Give brief summary of various objects

7.4 Mode, Type, and Class

Every `R` object has a mode, a type, and a class. You can interrogate an object for these properties with the functions `mode, typeof,` and `class`:

```
> m22 = matrix(0,2,2);   # m22 is a 2-by-2 matrix of zeros.
> mode(m22)    # Get or set mode of m22's elements.
  "numeric"    # Elements are numeric.
> typeof(m22)  # Of what type? (get only)
  "double"     # double (as opposed to integer).
> class(m22)   # Get or set class of object m22.
  "matrix"     # Class of m22 is matrix, uses matrix methods.
```

Note that the functions `mode` and `class` can also be assigned (set), to coerce the properties of an object:

```
> m22 = matrix(0,2,2)       # m22 contains numeric zeros.
> mode(m22) = "character"   # Coerce mode of m22.
  # Elements of m22 are now the character "0".
```

NOTE: a given coersion may not be possible because some coersions are not defined. Even when a coersion is possible, reversing the process may not be.

7.5 Creation, Coersion, and Detection

Each class has methods which include ways of creating, coercing to (where possible), and detecting its objects. For example, there are three functions for the class `"matrix"`:

```
   matrix   # Create a matrix.
as.matrix   # Coerce object to a matrix.
is.matrix   # Test if an object is a matrix
```

The following classes of object have similar trios of functions:

```
array   character  complex  data.frame  double
factor  list       logical  matrix      numeric
raw     ts (time series)    vector
```

NOTE: some entries in the above list actually represent several classes. For example, a matrix M might contain numbers:

```
mode(M) is "numeric", class(M) is "matrix",
```

or it might contain strings:

```
mode(M) is "character", class(M) is "matrix".
```

The two matrix classes are not the same (the rules of matrix algebra are only for numeric), yet `is.matrix(M)` will be `TRUE` in either case. Similarly, the function `vector` can create an object `Obj` of mode `"numeric"` or `"character"` or `"list"`, yet `is.vector(Obj)` will be `TRUE` in all three cases.

Coercion can also be done implicitly, by storing an element in a vector or array of a less inclusive type, e.g. a single character stored in a numeric vector (or array) will coerce the entire vector to character. Every number in the array will become a character representation of its value. The hierarchy of coercion is:

```
logical < integer < numeric < complex < character < list
```

You can override coercion using the function `I`, which produces a list-like object of class `"AsIs"`.

7.6 Assignment and Indexing Functions

Function Name	Example	Meaning
`‘<-‘`	`x <- 2`	assignment
	or `x =  2`	assignment
`‘<<-‘`	`x <<- 2`	super-assignment
`‘[‘`	`x[2] x[2,3]`	indexed source
`‘[<-‘`	`x[3] <- 5`	indexed target
`‘[[‘`	`F[[3]]`	indexed source list, get contents
`‘[[<-‘`	`F[[3]] <- x`	indexed target list, set contents

Every assignment and/or indexing step is done by a function whose name is listed in the first column above, enclosed in back-ticks. When `R` parses such steps, it uses the functional form, so that

```
F[[3]] <- x   becomes   ‘[[<-‘(F,3,x)
```

7.7 Arithmetic

Basic arithmetic is usually denoted using infix operators:

```
 Infix
Operator   Example     Result   Name
   +       2 + 3          5     addition
   -       5 - 2          3     subtraction
   *       7 * 8         56     multiplication
   /       9 / 4        2.25    division
   ^       2 ^ 3          8     exponentiation
   %/%     11 %/% 4       2     integer division.
   %%      11 %% 4        3     modulo or remainder
```

Each of these operators is a function that can be written in functional form, e.g.:

```
  11 %/% 4   becomes  `%/%`(11,4)
  Because the function name has special characters,
     it is enclosed in quotes.

Magnitude and Rounding Functions:
   abs(x)   floor(x)   ceiling(x)   trunc(x)
   round(x,digits=n)  signif(x,digits=n)

Logical Infix functions:
   Numeric or character input,  Logical results:
   <, <=, !=, ==, >=, >
   Logical input, logical results:
   !, &, &&, |, ||

See also:  xor, which.
```

7.8 String Functions

```
 Function    Purpose

 c           Create vectors of (e.g.) strings.
 charmatch   Find substring in string vector.
*gregexpr    Locate and count patterns.
*grep        Find string patterns in a string vector.
             Returns a numerical index.
*grepl       Same as grep, returns a logical index.
*gsub        Substitute one substring for others.
 match       Find occurences of strings (e.g.) in a vector.
 paste       Join several strings into one.
```

```
*regexpr     Locate patterns within strings and vectors.
*strsplit    Split a string into (e.g.) individual characters.
*sub         Substitute one substring for another.
 substr      Extract or replace substrings.
 substring   Extract or replace substrings (more versatile).
* These functions may use regular expressions in
  either POSIX or Perl format.
```

7.9 Elementary Functions

```
Trig functions:
   sin(x)   cos(x)   tan(x)    # input  in radians
Inverse Trig;
   asin(x)  acos(x)  atan(x)   # output in radians
   atan2(y,x)                  # input coordinates
Hyperbolic functions:
   sinh(x)  cosh(x)  tanh(x)
Inverse Hyperbolic:
   asinh(x)  acosh(x)  atanh(x)
Powers, Logarithms,  and Roots:
   exp(x)  log(x)  log10(x)  log2(x) sqrt(x)
   base^x  log(x,base)  # Powers and logs using any base.
Combinatorics:
   factorial(x)   choose(n,x)   gamma(x)   lgamma(x)
```

7.10 Continuous Probability Distributions

The `stats` package that comes with `R` provides many probability functions in four flavors signified by prefixes:

Prefix:	Meaning:
d	Probability density.
p	Cumulative probability.
q	Quantiles (inverse cumulative probability).
r	Random numbers from the distribution.

Each of these four prefixes is used by attaching it to a stem for the desired function, e.g. `dnorm, pcauchy, qhyper, rgamma`. Recall that help (?) may be requested only on a complete function name, not on a separate stem or prefix. The stems available are:

Stem	Distribution	Stem	Distribution
beta	beta	logis	logistic
binom	binomial	nbinom	negative binomial

cauchy	Cauchy	norm	normal
chisq	chi-squared	pois	Poisson
exp	exponential	signrank	Wilcoxon signed rank
f	Fisher's F	t	Student's t
gamma	gamma	unif	uniform
geom	geometric	weibull	Weibull
hyper	hyperbolic	wilcox	Wilcoxon rank sum
lnorm	lognormal		

```
See also:  sample and sample.int for sampling
           values from a (possibly arbitrary) vector.
```

7.11 Vector Functions

Many functions that accept scalar arguments will also accept vector arguments. The above lists of infix operators and elementary functions also apply to vectors, with repeat rules to account for length mismatches. Other functions are designed to create special vectors:

```
x = 0:100         # x = vector of 101 integers.
y = .02*pi*x      # y = 0 to 2*pi, 101 values.
z = sin(y)        # z = 1 period of a sin curve, 101 values.
x = seq(0,2,len=5)
                  # x = 0 to 2, 5 values.
y = rep(x,each=2,len=20)
                  # y = 0 0 .5 .5 1 1 1.5 1.5 2 2
                  #     and repeat until length(y) is 20.
# length() will get or set the number of elements.
LenY = length(y)  # Get length(y), which is 20.
length(y) = 5     # Set length(y) to 5, drop elements 6-20.
```

7.12 Matrix and Array Functions

M is a matrix. A is an array:

Function	Use
matrix	Create a matrix
as.matrix	Coerce non-matrix object to matrix
is.matrix	Test if object is of class matrix
array	Create and array
as.array	Coerce non-array object to class array
is.array	Test if object is of class array
dim	Get or set array dimensions
rownames	Get or set matrix row names

`colnames`	Get or set matrix column names
`dimnames`	Get or set array dimension names
`A[iv,jv,kv]`	Get or set subarray of A, where iv, jv, and kv are numeric vectors or dimension name vectors.
`det(M)`	Determinant of M
`t(M)`	Transpose of M (conjugate transpose for complex data)

7.13 Functions that Apply Other Functions

Some functions are designed to apply other functions to specific kinds of data. The names of some of these functions end in the word `apply`:

Function	Data
`apply`	Matrices, arrays
`lapply`	Lists
`sapply`	vectors
`tapply`	tables
`with`	data frames
`by`	data frames
`aggregate`	data frames

7.14 Functions that Sort

Function	Purpose
`sort`	Sort a vector, increasing or decreasing order.
`order`	Provide a permutation index that sorts.
`rev`	Reverse the order of a vector.
`merge`	Merge two dataframes.
`levels`	Get or set an ordering rule for factors that tells how levels are to be sorted in the future.

Chapter 8

Dataframes

8.1 What is a dataframe?

A dataframe (DF) is a named list that can be thought of as a collection of columns. A column's name must not contain blanks, e.g. use "Hair.Color" rather than "Hair Color". Each column records a distinct kind of data, e.g. in the dataframe called `CompPrice` (see example below): price is in column 1, item name in column 2, store name in column 3, and city name in column 4. In this case column 1 is numeric, while columns 2, 3, and 4 are factor. Columns of strings are coerced to factor by default. There are several kinds of data that a column can contain: e.g. numbers, factors, strings, logicals, and dates, but within each column, consistency of class is the rule.

All columns must be the same length. Each row of a DF is a particular case, e.g. row 1 in the `CompPrice` example below tells the following 4 pieces of information:

```
  col 1      col 2      col 3            col 4
The price   of bread   at Trader Ted's   in Miami
```

If some datum for a particular case is not available, its entry should contain the value NA.

8.2 Example of a Non-dataframe

To illustrate what a DF is not, recall an example from Chapter 4, in which we set up a 3-by-4-by-6 array, called `A346`, of prices for 3 items in 4 store chains in 6 cities. We denoted which items, store chains, and cities were being priced by naming the rows, columns, and pages as follows:

```
     dimnames(A346) = list(   # a list of string vectors.
# Row names first:
     c("Bread","Detergent","Eggs"),
# Column names second:
     c("Cheap.Gourmet","McDee's","Trader.Ted's","WallMark"),
# Page names third:
     c("Miami","Atlanta","Washington",
       "Philadelphia","New York","Boston"))
```

Each price resides in some row, column, and page, the names of which tell what item, store, and city the price is from. This 3-dimensional object is not a DF because all the prices (e.g.) do not reside in the same column.

8.3 Changing to a Dataframe Format

To change `A346` to a DF, create four vectors named `Prices`, `Items`, `Stores`, and `Cities`, all of length `3x4x6` or `72`. Put the 72 prices from `A346` in the `Prices` vector, put the item names in the `Items` vector, repeated as often as necessary to make 72, and so on as follows:

```
# Prepare vectors of data.
     Prices = as.vector(A346);
     Items  = rep(dimnames(A346)[[1]],len=72)
     Stores = rep(dimnames(A346)[[2]],each=3,len=72)
     Cities = rep(dimnames(A346)[[3]],each=12,len=72)

# Create a dataframe from these 4 vectors.
     CompPrice = data.frame(Price=Prices,
        Item=Items, Store=Stores, City=Cities)

# Order the cities from south to north
#   using 3rd-dimension names from A346 above, with
#   some additional city names for future entry.
    CompPrice[,4] = factor(CompPrice[,4],
       ordered=TRUE, levels=c(
       "Miami","Atlanta","Raleigh","Richmond",
       "Washington","Baltimore","Philadelphia",
       "New York","Hartford","Boston"))
```

The `levels` clause above does not reorder the cities as they appear in the factor `City`. It provides an order for future sorting, if and when such sorting is requested. Notice also that we have introduced more city names in the `levels` clause than there are in the `City` factor itself. Again, this does not force new entries into the factor, but simply provides permission and an implicit ordering for future entries from these (as yet unsampled) cities. You will find that DFs can be reluctant to accept new and totally unanticipated factor levels.

8.4 Dataframe Input

DFs can exist in files, which can then be read:

```
FilePath = "c:\\temp\\worms.txt"
worms = read.table( FilePath, header=TRUE )
```

A header, if there is one, is on the first line of the file. Since the columns are named, it is the header's job to exhibit those names. But beware: if the names in the header contain blanks, each blank-delimited string will be counted as a separate name, causing a mismatch between names and columns. Edit the file if necessary to replace blanks within column names with dots.

8.5 Accessing Parts of Dataframes

DFs look like matrices, in that they are a strictly rectangular arrangement of information. Unlike numerical matrices, the type of information can vary from column to column, but the indexing schemes of the two object classes is quite similar:

```
# Example 1:  df = 1st 3 rows of CompPrice
     df = CompPrice[1:3,]
# Example 2:  df = all entries for city of Miami.
     df = CompPrice[CompPrice$City == "Miami",]
# Make columns separately accessible.
     attach(CompPrice)
# Example 3:  Now Ex. 2 can be shorter.
     df = CompPrice[City == "Miami",]
# Example 4:  df = rows for which
#    prices are > mean price.
     df = CompPrice[Price>mean(Price),]
# Example 5:  Get all prices except WallMark's.
     v = CompPrice$Price[Store != "WallMark"]
```

Examples 1-4 above produce DFs as a result. Such examples retain column names, and row names too, if there are any. But the result of extracting rows and columns from an existing DF might not be another DF. Example 5 produces a numeric vector, and in general, extracting data from a single column of a DF will produce an object with the class of that column, just as the `attach` function does.

8.6 Deleting Parts of Dataframes

As with matrices, indexes can be used to remove cross-sections (i.e. rows or columns) from DFs:

```
# DF is a dataframe with all rows pertaining to
# WallMark Stores deleted from CompPrice:
     DF = CompPrice[Store != "WallMark",]
# Remove the 1st 6 rows from DF:
     DF = DF{-(1:6),]
# DF =  CompPrice with the "City" (4th) column removed:
     DF = CompPrice[,-4]
```

8.7 Appending to Dataframes

The functions `rbind` (append rows) and cbind (append columns), may be used to expand DFs. These functions are versatile enough to recognize how to accomplish this even when the object being appended is not itself a DF:

```
# DF1 & DF2 are dataframes with matching column names.
     DF3 = rbind(DF1,DF2)
# DF3 has the rows of DF1 followed by the rows of DF2.
# If the order of column names in DF1 & DF2 were different,
#  they will be properly matched,
#  with the order in DF1 taking precedence.
# Dataframes can be taken apart and reassembled:
     DF1 = CompPrice[,1:2]   # Cols 1&2
     DF2 = CompPrice[,3:4]   # Cols 3&4
     CP  = cbind(DF1,DF2)    # CP is CompPrice reassembled.
# Profit is a vector of profits for each item
#  in each store in each city, 72 entries.
     CompPrice = cbind(CompPrice,Profit)
# CompPrice now has a 5th column named "Profit".
```

8.8 Sorting Dataframes

Sorting in `R` is mainly done by two functions: `sort` and `order`. The function `sort`, the direct method, puts data in ascending or descending order. The function `order` produces a vector of permuted (shuffled) indexes which can be used in turn to sort the original data and related data as well:

```
v = c(22,44,33,11)          # Unsorted data.
vincr = sort(v)             # vincr = c(11,22,33,44)
vdecr = sort(v,dec=TRUE)    # vdecr = c(44,33,22,11)
iv = order(v)               # iv = c(4,1,3,2)
vord = v[iv]                # vord = c(11,22,33,44)
```

Note how `sort(v)` and `v[order(v)]` yield the same result. If producing a sorted version of `v` were the only goal, `sort(v)` would be preferable. But in a DF context, we often want to permute lots of related data along with the vector in question, so `order` becomes the function of choice.

The function `order` can operate on several columns simultaneously, by passing several arguments:

```
# Continuing the CompPrice example,
#    sort the CompPrice dataframe giving
#    the Item column highest priority,
#    then the Store column, and finally City.
# Items will vary slowest, cities fastest.
```

```
ir = order(Item,Store,City) # ir is the order vector.
CompPrice2 = CompPrice[ir,] # Sort rows of CompPrice.
```

In reshuffling the rows, ties (equal entries) in the ”Item” column will be broken by the ”Store” column, and ties in both columns will be broken by the ”City” column. When all three columns are tied, rows will be arranged in the order they appeared in the original DF.

8.9 Merging Dataframes

DFs can be merged using the function `merge`. This function can easily produce what you don't expect, but considering the variety of tasks it is expected to perform, it begins to seem reasonable.

8.9.1 A Strange Example

For example, consider the seemingly simple task of merging two vectors into a dataframe:

```
v1 = c(3,5,1);   v2 = c(6,2,4)  # 2 data vectors
z = merge(v1,v2);               # Merge v1 & v2.
# The following is the resulting DF with columns
#    printed as rows to save lines:
x:   3  5  1  3  5  1  3  5  1
y:   6  6  6  2  2  2  4  4  4
```

Where we might expect a single sorted vector of 6 values, `x` is instead a 9-row DF, with values from `v1` in column 1 (named x) and values from v2 in column 2 (named y), unsorted but repeated to make a cross-product pattern.

8.9.2 The Use of rbind

The most intuitive use of `merge` is with two DFs (call them DF1 and DF2) that have identical column names. Using the function `rbind` to join two DFs DF1 and DF2 as in:

```
DF3 = rbind(DF1,DF2)
```

`rbind` will try to put the rows of DF2 under the those of DF1. There will be no attempt to sort anything. If the columns of DF2 have the same names as those in DF1, but in a different order, the columns from DF2 will be reordered to conform to DF1. However, this attempt will fail if DF2 contains factors with levels that are not anticipated by DF1. Every factor level in DF2 must be known by DF1 either because it actually appears in DF1 or because it was included in the `levels` clause when the DF1 factor was defined. For an example of how to provide levels, see the code in the section ”What is a Dataframe?” where the DF `CompPrice` is created from the array `A346`.

8.9.3 The Use of merge

In the examples that follow, the clause `all=TRUE` will appear in every call to the `merge` function. This means that every line from the two merged DFs must appear in the result. What happens when this clause is omitted is left for the reader to explore.

By comparison to `rbind`, the `merge` function will join the two DFs and account for positions of columns, as `rbind` did. However, unlike `rbind`, `merge` will sort the rows:

```
# Create two new dataframes DF1 & DF2 by extracting data
#    for Miami and Boston from CompPrice:
# Also in DF1, reverse order of columns from CompPrice,
#    i.e., Columns of DF1 are:
#    "City", "Store", "Item", and "Price".
   DF1 = CompPrice[CompPrice$City=="Miami" ,c(4,3,2,1)]
   DF2 = CompPrice[CompPrice$City=="Boston",]

# ir1 & ir2 are 2 random reorderings of the integers 1:12:
   ir1=sample.int(12);   ir2=sample.int(12)
# Shuffle the rows of DF1 & DF2:
    DF1=DF1[ir1,];   DF2=DF2[ir2,]

# Merge DF1 & DF2:
   DFboth = merge(DF1,DF2,all=TRUE)
```

Here are some salient points about the above example:

1. Because there are 12 prices from each city, `DF1` and `DF2` will each contain 12 rows (3 items times 4 stores), but the `City` column will have only one city name (`"Miami"` or `"Boston"`) repeated 12 times. Despite each `City` factor having only one level, all of the original levels are still present in the background, as you can check by e.g. `levels(DF1$City)`.

2. The index `c(4,3,2,1)` in the definition of `DF1` reverses the order of columns from `CompPrice`, with names and properties intact.

3. Each invocation of `sample.int` produces an independent reordering if the integers, so even though the definitions of `ir1` and `ir2` look identical, they're not. As a result, the rows of `DF1` and `DF2` will not be shuffled in the same way.

4. Since the order of columns does not match between `DF1` and `DF2`, `merge` reorders the columns from DF2 when creating DFboth. Finally, `merge` sorts the rows of the composite DFboth without regard to which DF they came from.

8.9.4 Controlling the Sorting Order

It remains to specify what is meant by sorting the rows. In the section "Sorting Dataframes", we saw that the `by` parameter can be used to give the columns any priority we choose in determining the sort. A similar device can be used in merging. By default, column priority is the order of the columns in DF1. The columns of DF2 are shuffled accordingly, as long as DF1 and DF2 share the same column names. If we wish to change the priority, we can set the by.x parameter, which reorders the columns from DF1 into DFboth:

```
# DF1 has 5 columns:
#   "City", "Store", "Item", "Price", and "Profit".
# DF2's columns have the same names, different order:
#   "Price", "Item", "Store", "City", and "Profit".
# Merge DF1 & DF2, but specify the priorities:
DFboth = merge(DF1,DF2,all=TRUE,
          by.x=c("Item","Store","City","Price","Profit"))
```

In the composite DFboth, items are in the first column and have the highest priority, stores are in the next column, and so on. Now, suppose DF2 has columns that correspond to DF1, but with different order and names. We not only specify the priorities using terminology from DF1, but we also must specify the correspondence between columns of DF1 and DF2

```
# Here are corresponding column names of DF1 & DF2:
#    DF1 columns:   Price  Item   Store   City  Profit
#    DF2 columns:   Cost   Goods  Outlet  Town  Markup
DFboth = merge(DF1,DF2,all=TRUE,
   by.x=c("Item",  "Store",  "City", "Price", "Profit"),
   by.y=c("Goods", "Outlet", "Town", "Cost",  "Markup"))
```

Once again, items will have the highest sorting priority and so on.

8.9.5 Non-matching Columns

So far, we have merged two DFs where every column in one corresponds to some column in the other. Now let us consider the case where some columns of one DF have no counterparts in the other:

```
# c1 & c2 are corresponding in both Dfs,
# but c3 is only in DF1 and c4 is only in DF2
# with no common meaning between them.
DF1 = data.frame(c1 = c( 5,  1,  3),    # c1 also in DF2
                 c2 = c("c","e","a"),   # c2 also in DF2
                 c3 = c("x","y","z"))   # c3  not in DF2
DF2 = data.frame(c1 = c( 4,  6,  2),    # c1 also in DF1
                 c2 = c("f","d","b"),   # c2 also in DF1
                 c4 = c("t","u","v"))   # c4  not in DF1
DFboth = merge(DF1,DF2,all=TRUE,        # Merge DF1 & DF2,
         by.x=c("c1","c2"),             #  specify
         by.y=c("c1","c2"))             #  correspondence.
```

As requested by `all=TRUE`, all rows from DF1 and DF2 will appear somewhere in DFboth:

```
  c1 c2 c3 c4
1  1  e  y NA
2  2  b NA  v
3  3  a  z NA
4  4  f NA  t
5  5  c  x NA
6  6  d NA  u
```

However, the column c3 in DFboth received no entries from DF2 (which has no such column), and likewise, the column c4 received no entries from DF1. Therefore in DFboth, those 6 non-entries are marked NA or "not available".

A fuller calling sequence of merge, with default values, is:

```
DFboth = merge(x,y,      # the 2 DFs to be merged
     by = intersect(names(x),names(y)), # Order key
     by.x = by,   # Order key, another way
     by.y = by,   # Corresponding columns of y
     all=FALSE,   # Include all rows?
     all.x=all,   # Include all rows of x?
     all.y=all,   # Include all rows of y?
     sort=TRUE)   # Sort the result?
```

The complete set of options can be seen by typing ?merge.